Mathématiques Appliquées
MAP552 - TP Initiation Python **Aldjia Mazari**

Travaux Pratiques

1. Programmation matricielle

Eviter les boucles en programmant "matriciellement".

En `python`, la programmation matricielle proposée dans `numpy` est plus rapide que celle reposant sur des boucles. Le programme suivant compare le calcul d'une valeur approchée de la constante d'Euler $\gamma \approx \sum_{i=1}^{n} \frac{1}{i} - \ln(n)$ pour $n \gg 1$ en programmation matricielle et via une boucle. Quel est le facteur de temps de calcul entre les deux méthodes ?

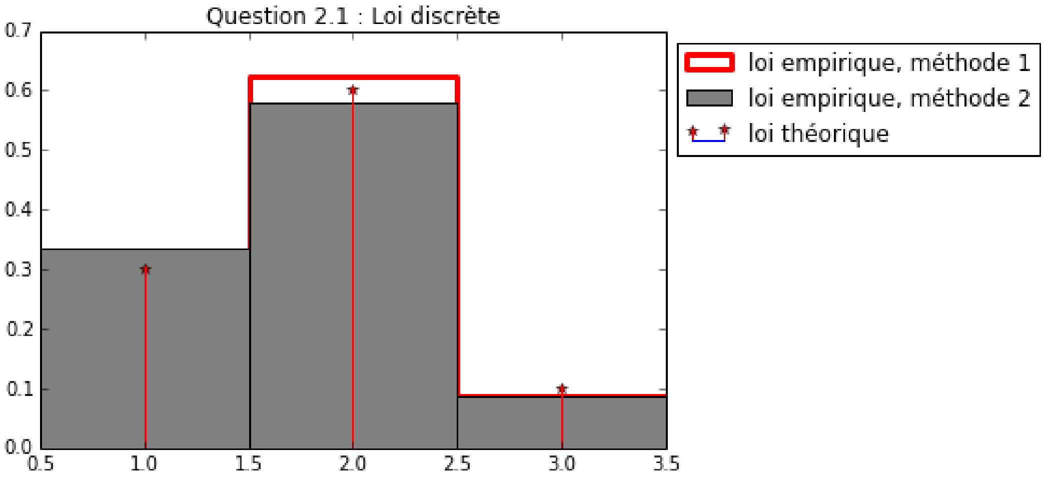
```
# Approximation de la constante d'Euler, via une boucle
# VS via l'utilisation des fonctions cablees
#
import numpy as np
import time
#import time pour avoir la fonction qui donne le temps CPU
# different de la fonction time de base, qui donne l'heure

n = 10**6
# Methode 1. Boucle for
print "-" * 40
print "Methode 1. Boucle for"
print "-" * 40
t1 = time.time()
x = 0
for i in range(1, n+1):
    x += 1. / i
print "gamma=", np.sum(x) - np.log(n)
t2 = time.time()
print "Cela a pris ", t2 - t1, " secondes"
# Methode 2. Numpy, programmation vectorielle
print "-" * 40
print "Methode 2. Numpy, programmation vectorielle"
t1 = time.time()
print "gamma=", np.sum(1. / np.arange(1, n+1)) - np.log(n)
t2 = time.time()
print "Cela a pris ", t2 - t1, " secondes"
print "-" * 40
# Exemple de resultats
#----------------------------------------
#Methode 1. Boucle for
#----------------------------------------
#gamma= 0.577216164901
#Cela a pris 0.217562913895 secondes
#----------------------------------------
#Methode 2. Numpy, programmation vectorielle
#gamma= 0.577216164901
#Cela a pris 0.0111699104309 secondes
#----------------------------------------#
```

2. Simulation de variables aléatoires discrètes

2.1. **Un exemple simple de loi discrète.** On considère la loi P sur l'ensemble $\{1, 2, 3\}$ donnée par les probabilités $P(\{1\}) = 0.3$, $P(\{2\}) = 0.6$, $P(\{3\}) = 0.1$. Simulez un grand nombre de v.a. i.i.d. de loi P, en faire l'histogramme à l'aide des fonctions `bincount` et `bar` et comparer le résultat obtenu avec la représentation en bâtons, sur le même graphique, de la loi P (que l'on obtient à l'aide de la fonction `stem`).

```
# -*- coding: utf-8 -*-
import numpy as np #import de numpy sous l'alias np
import numpy.random as npr #import de numpy.random sous l'alias npr
```

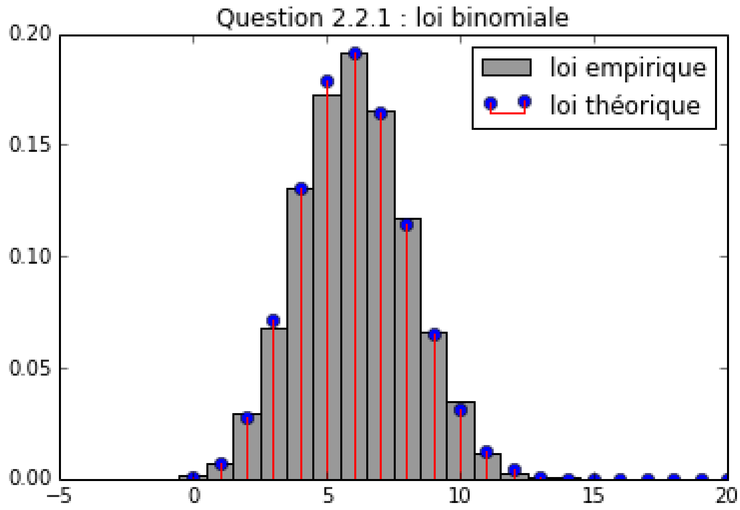
```
import matplotlib.pyplot as plt #import de matplotlib.pyplot sous l'alias plt
################################################################################
## Question 2.1
################################################################################
## Pour cette question, donnons deux facons de faire:
##  1. Une methode "a la main" ou on part de tirages de loi uniforme sur [0, 1]
##  2. Une fonction toute faite de numpy
# Nombre de simulations
n = 1000
# Methode 1. On utilise la fonction numpy.random.rand pour effectuer le tirage
# d'un echantillon X de taille n, suivant une loi uniforme sur [0,1]
X = npr.rand(n)
# On construit le vecteur M1 de taille X,tel que, quelque soit i
# M1(i) = 1 si X(i) <= 0.3
# M1(i) = 2 si 0.3 < X(i) <= 0.9
# M1(i) = 3 si X(i) > 0.9
M1 = 1 * (X <= 0.3) + 2 * np.logical_and(0.3 < X, X <= 0.9) + 3 * (X > 0.9)
# Methode 2 : on utilise la fonction choice de numpy.random qui genere directement
# un echantillon de taille n, prenant les valeurs donnees en premier parametre
# selon les probabilites donnees par le parametre p
valeurs = np.array([1, 2, 3])
probas = np.array([0.3, 0.6, 0.1])
M2 = npr.choice(valeurs, size=n, p=probas)
#
# On dessine les resultats
# Remarque : la fonction hist de matplotlib n'est pas tres appropriee pour
#           afficher l'histogramme d'une loi discrete. On va faire autrement.
# On compte le nombre de fois que l'on voit 1, 2, 3 dans le vecteur
#
# numpy.bincount compte, dans un tableau d'entiers positifs ou nuls,
# le nombre d'elements du vecteur egaux a 0, 1, 2, etc.
# On cree donc le vecteur count1, via np.bincount(M1), dont on enleve la premiere valeur
# qui traduit que l'on observe 0 fois 0, d'ou count1 = np.bincount(M1)[1:], idem pour
# la seconde methode avec le vecteur M2
counts1 = np.bincount(M1)[1:]
counts2 = np.bincount(M2)[1:]
# On convertit le vecteur counts1 en float pour pouvoir diviser par n
counts1 = np.array(counts1, dtype=float)
counts1 /= n
# On peut aussi diviser par float(n) pour ne pas faire une division entiere
counts2 = np.array(counts2, dtype=float)
counts2 /= n
#
# Affichage des resultats
print "-" * 40
print "Question 2.1"
print "-" * 40
# On affiche l'histogramme
plt.bar(valeurs - 0.5, counts1, width=1., label=u"loi empirique, méthode 1",edgecolor='r',color='',
    linewidth=3)
plt.bar(valeurs - 0.5, counts2, width=1., label=u"loi empirique, méthode 2",color=[.5,.5,.5]) #codage
    rgb de la couleur des barres
# On affiche la loi theorique pour comparer
plt.stem(valeurs, probas, label=u"loi théorique",linefmt='r-',markerfmt='r*',basefmt='b-')
# On dessine la legende
plt.legend(loc='upper left' , bbox_to_anchor = (1., 1.)) #on place la legende en haut a droite de la
    figure
# On donne un titre
plt.title(u"Question 2.1 : Loi discrète")
# On affiche toutes les figures
plt.show()
```



2.2. **Les lois discrètes classiques.**

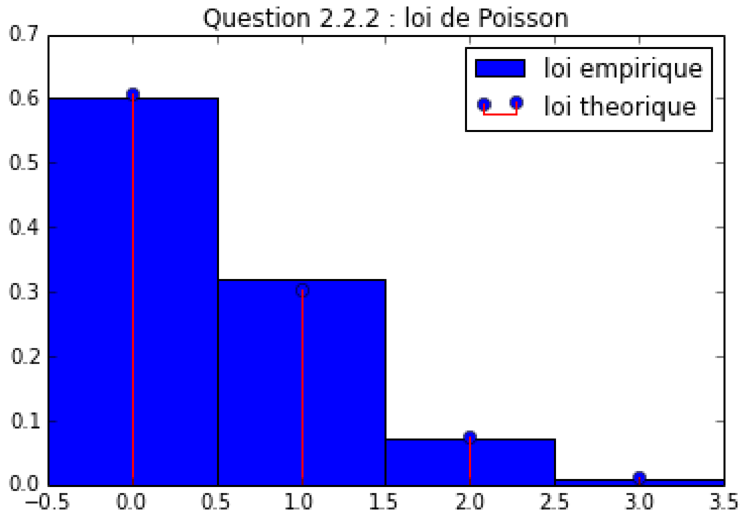
(1) On considère $B(n,p)$ la loi binomiale de paramètres n,p. Ecrire une procédure dans laquelle on simule un grand nombre de v.a. de loi $B(n,p)$ (avec la fonction `binomial` de `numpy.random`), faire l'histogramme de l'échantillon obtenu et comparer avec la représentation en bâtons, sur le même graphique, de la loi $B(n,p)$ (que l'on obtient à l'aide de la fonction `binom.pmf` de `scipy.stats`).

```
# -*- coding: utf-8 -*-
import numpy as np
import numpy.random as npr
import scipy.stats as sps
import matplotlib.pyplot as plt
############################################################################
## Question 2.2.1
############################################################################
#Nombre de tirages
N = 10000
#Parametres de la binomiales
n = 20
p = 0.3
#N tirages de la binomiale
B = npr.binomial(n, p, N)
#Poids de la binomiale aux points 0, ...., n
valeurs = np.arange(n+1)
f = sps.binom.pmf(valeurs, n, p)


#Histogramme de la distribution empirique avec bincount et bar
partial_counts = np.bincount(B)
#on complete avec des zeros le vecteur de comptage en un vecteur de meme taille que valeur
counts = np.zeros(n+1)
counts[0:len(partial_counts)] = partial_counts
counts = np.array(counts, dtype=float) # on transforme le vecteur d'entiers en vecteur de
    flotants
counts /= N
plt.bar(valeurs - 0.5, counts, width=1., label="loi empirique",color=[.6,.6,.6]) # les barres
    seront grises
#
#distribution theorique
plt.title("Question 2.2.1 : loi binomiale")
plt.stem(valeurs, f, "r", label=u"loi théorique")
plt.legend()
plt.show()
```



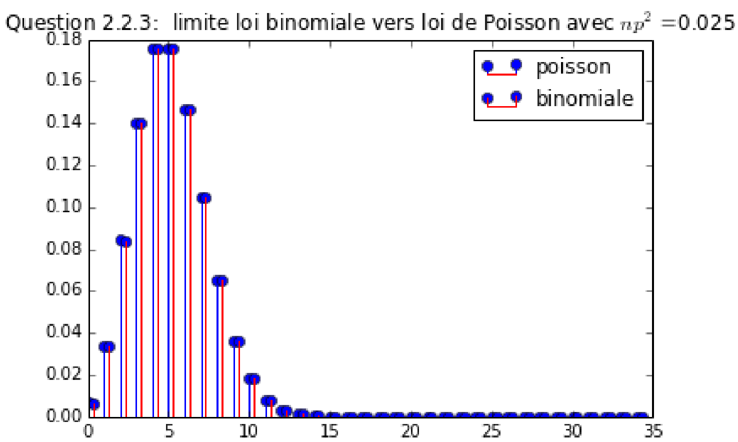
(2) Faire de même avec une loi de Poisson (on utilisera la fonction `poisson` de `numpy.random` et la fonction `poisson.pmf` de `scipy.stats`).

```
# -*- coding: utf-8 -*-
import numpy as np
import numpy.random as npr
import scipy.stats as sps
import matplotlib.pyplot as plt
################################################################################
## Question 2.2.2
################################################################################
# Intensite de la loi de poisson
mu = 0.5
# Nombre de simulations
N = 1000
# Tirage d'un echantillon de loi de Poisson, de taille N
X = npr.poisson(mu, N)
n = 20
# numpy.bincount compte, dans un tableau d'entiers positifs ou nuls,
# le nombre d'elements du vecteur egaux a 0, 1, 2, etc.
# on divise le resultat par float(N), pour ne pas faire une division entiere
#
counts = np.bincount(X) / float(N)
#
# Discretisation de la loi theorique
#
x = np.arange(len(counts))
f_x = sps.poisson.pmf(x, mu)
#
# Affichage compare, empirique vs theorique
#
plt.bar(x - 0.5, counts, width=1., label="loi empirique",edgecolor=[.6,.6,.6],color='')
# Nota ; edgecolor=[.6,.6,.6],color='' => le contour des barres sera gris, les barres ne
    seront pas remplies
p2 = plt.stem(x, f_x, "r", label=u"loi théorique")
plt.title("Question 2.2.2 : loi de Poisson")
plt.legend()
plt.show
```



(3) Un calcul simple montre que lorsque n tend vers l'infini, la loi $B(n, \mu/n)$ tend vers la loi de Poisson de paramètre μ. En pratique, on assimile $B(n, p)$ à la loi de Poisson de paramètre np dès que $np^2 < 0.1$. Illustrer cette *proximité de lois* en affichant, sur le même graphique, leurs histogrammes en bâtons (sans faire aucun tirage).

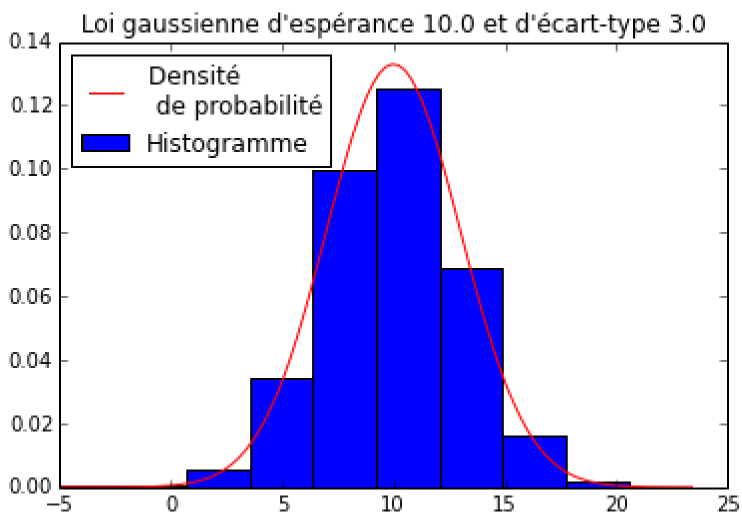
```
# -*- coding: utf-8 -*-
import numpy as np
import scipy.stats as sps
import matplotlib.pyplot as plt
##############################################################################
## Question 2.2.3
##############################################################################
n = 1000
p = 0.005
# Intensite de la loi de poisson
mu = p * n
np2 = n * p ** 2
x = np.arange(0, 7 * mu)
# Simulation de la loi de Poisson
p_poisson = sps.poisson.pmf(x, mu)
# Simulation de la loi Binomiale
p_binom = sps.binom.pmf(x, n, p)
#
#Affichage compare des resultats
plt.stem(x, p_poisson, markerfmt='*',label="poisson")
#On shifte de 0.3 en abscisse les resultats obtenus pour la loi binomiale
plt.stem(x+0.3, p_binom, "r", label="binomiale")
plt.title("Question 2.2.3: limite loi binomiale vers loi de Poisson avec $np^2=$"+str(np2))
plt.legend()
plt.show()
```



3. Simulation de variables aléatoires continues

(1) Simuler un grand nombre de variables aléatoires gaussiennes standard, représenter l'histogramme associé et comparer à la densité gaussienne. On utilisera les fonctions `randn`, `linspace`, `hist`, `plot` et la fonction `norm.pdf` de la librairie `scipy.stats`

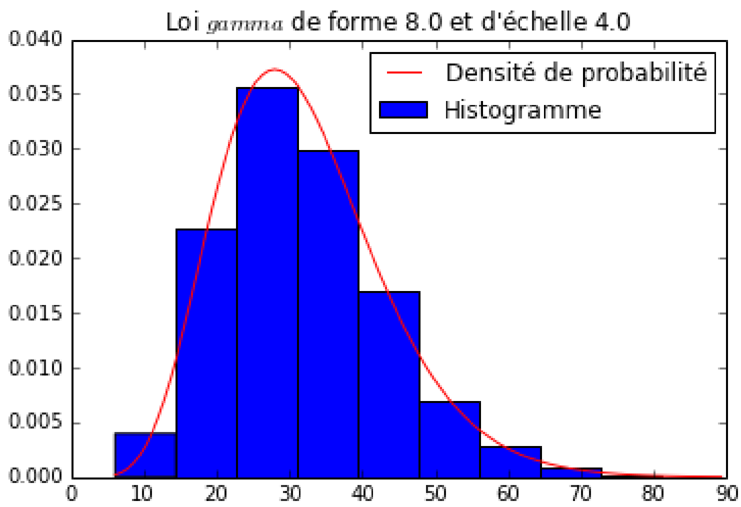
```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as npr
import scipy.stats as sps

#nombre de tirages
N = 10**6
# Esperance
m = 10.
# Ecart-type
s = 3.
#tirages
X = m + s * npr.randn(N)
x = np.linspace(min(X), max(X), 100)
#densite
f_x = sps.norm.pdf(x,m,s)
#figure()
plt.hist(X, normed=True, label="Histogramme")
plt.plot(x, f_x, "r", label=u"Densité \n de probabilité")# Noter le retour a la ligne dans la
    legende
plt.legend(loc=2)
plt.title(u"Loi gaussienne d'espérance " + str(m) + u" et d'écart-type " + str(s))
plt.show()
```



(2) Faire de même avec des variables aléatoires de loi *gamma*. On utilisera la fonction `gamma` de `numpy.random` et la fonction `gamma.pdf` de `scipy.stats`

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as npr
import scipy.stats as sps

#nombre de tirages
N = 5000
#parametre de la loi gamma, k=shape et theta=scale
shape= 8.
scale= 4.
#tirages
X = npr.gamma(shape, scale,size= N)
```

```
#densite
x = np.linspace(min(X), max(X), 100)
f_x = sps.gamma.pdf(x, shape, scale=scale)
#figure()
plt.hist(X, normed=True, label="Histogramme")
plt.plot(x, f_x, "r", label=u"Densité de probabilité")
plt.legend()
plt.title("Loi $gamma$ de forme " + str(shape) + u" et d'échelle " + str(scale))
plt.show()
```

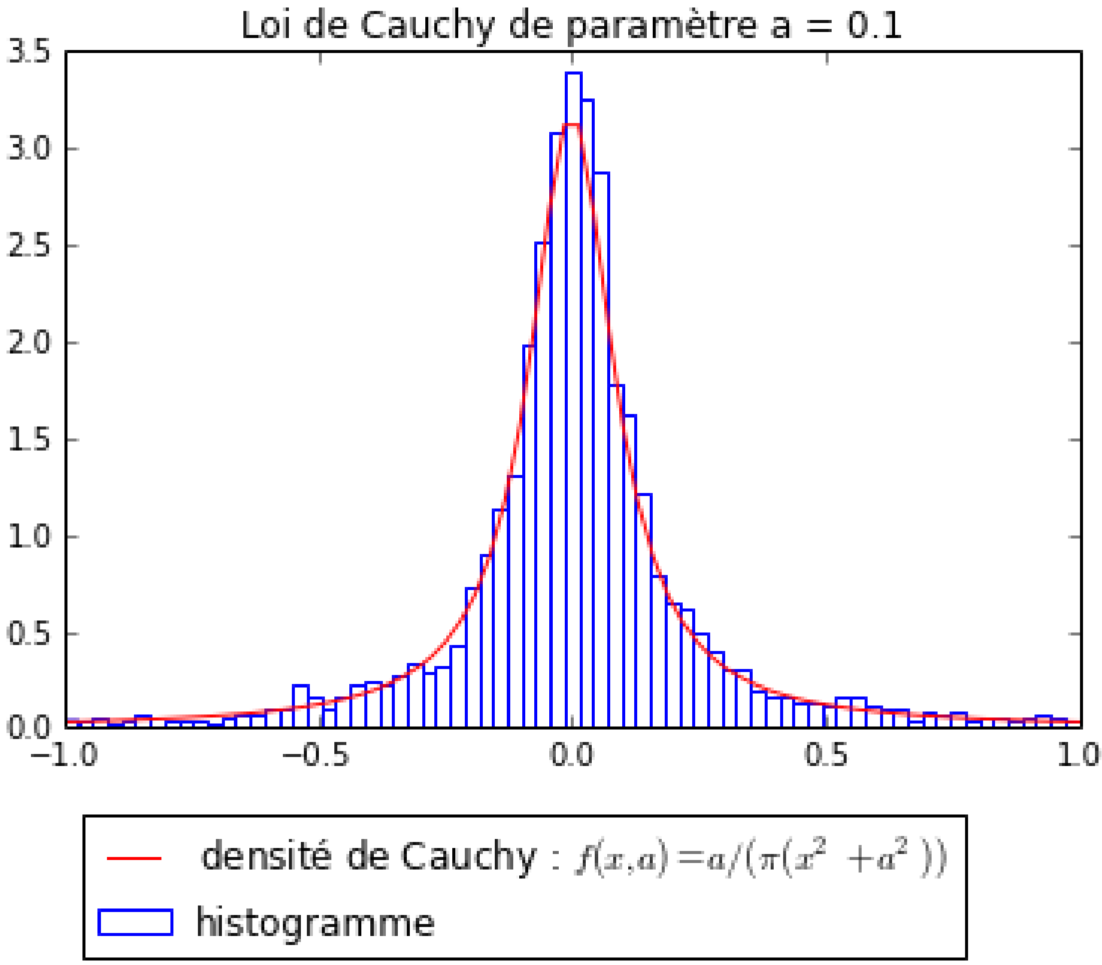


(3) **inversion de la fonction de répartition**

La *loi de Cauchy de paramètre* a est la loi de densité $f_a(x) = \dfrac{1}{\pi} \dfrac{a}{x^2 + a^2}$ sur $\mathbb{R}$. Simuler un grand nombre de variables aléatoires de loi de Cauchy de paramètre a, représenter l'histogramme associé et le comparer à la densité. Que se passe-t-il lorsque a s'approche de 0 ? *Rappel : On rappelle que si* U *est une v.a. uniforme sur* $]0,1[$ *et* F *la fonction de répartition d'une loi* μ*, alors* $F^{-1}(U)$ *est distribuée selon* μ.

```
# -*- coding: utf-8 -*-
from pylab import pi # import de la constante pi
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as npr

#densite de la loi de Cauchy
def f(x, a):
    '''
    Calcul de la densite de la loi de Cauchy \n
    f(x,a) = a / (pi * (x^2) + a^2) \n
    x et a sont les parametres d'entree
    '''
    return a / (pi * (x ** 2 + a ** 2))
# fin definition de la fonction f
#inverse de la fct de repartition de la loi de Cauchy
def G(x, a):
    '''
    Calcul de l'inverse de la fonction repartition de la loi de Cauchy \n
    G(x,a) = a * arctan(pi*(x-0.5) \n
    x et a sont les parametres d'entree
    '''
    return a * np.tan(pi * (x - .5))
# fin definition de la fonction G
#nombre de tirages
N = 5000
#parametre de la Cauchy
a =0.1
# Borne superieure de l'intervalle de discretiation
bound=10*a
```

```
#Simulation d'un ecahntillon d'une loi uniforme sur [0,1], de taille N
U = npr.rand(N)
# Calcul de l'inverse de la fonction repartition de la loi de Cauchy pour chaque composante de
    U
X = G(U, a)
# Discretisation de l'intervalle [-bound,+bound]
x = np.linspace(-bound, bound, int(np.sqrt(N)))
#Calcul de la densite de la loi de Cauchy pour chaque composante de x
y = f(x, a)
# Affichage des resultats
plt.figure()
plt.hist(X, normed=True, bins=round(np.sqrt(N)), label="histogramme", range=(-bound, bound),
    edgecolor='b',color=[1,1,1])
legende1 = u"densitÃ© de Cauchy : "+r"$f(x,a) = a/(\pi(x^2+a^2))$"
plt.plot(x, y, "r", label=legende1)
plt.legend(loc='upper left' , bbox_to_anchor = (0., -.1))
plt.title(u"Loi de Cauchy de paramÃ¨tre a = " + str(a))
plt.show()
```



(4) **méthode de rejet**

La *loi de Wigner* est la loi de support $[-2, 2]$ et de densité $\frac{1}{2\pi}\sqrt{4-x^2}$. Simuler un grand nombre de variables aléatoires de loi de Wigner, représenter l'histogramme associé et le comparer à la densité.

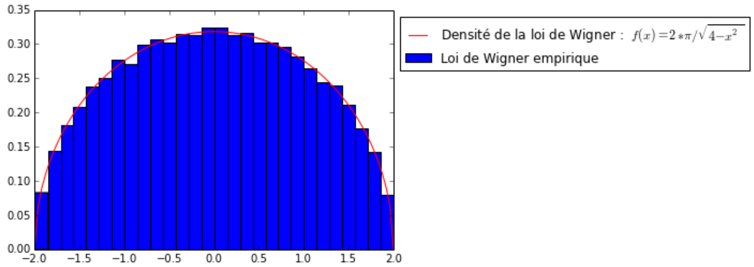
Rappel : Pour simuler une v.a. de densité f de support $[a, b]$ et telle que $0 \leq f \leq M$, on utilise une suite $(U_n, V_n)_{n\geq 1}$ de couples indépendants de variables aléatoires telles que pour tout n, U_n, V_n sont indépendantes et de lois uniformes sur respectivement $[a, b]$, $[0, M]$, on pose

$$\tau = \min\{n \geq 1\,;\, V_n \leq f(U_n)\},$$

et la variable aléatoire U_τ suit alors la loi de densité f. C'est la méthode de simulation par rejet. Notons que plus M est petit, plus cette méthode est rapide, on a donc intérêt à choisir $M = \|f\|_\infty$.

```
# -*- coding: utf-8 -*-
```

```
from pylab import pi
from time import time
import numpy as np #import de numpy sous l'alias np
import numpy.random as npr #import de numpy.random sous l'alias npr
import matplotlib.pyplot as plt #import de matplotlib.pyplot sous l'alias plt

#Nombre de tirages
N = 10**5
#on genere dans X des realisations de la loi de Wigner par methode du rejet
borne = 1. / pi
temps_debut = time()
u=4*npr.rand(N)-2 # u = tirage de N valeurs suivant une loi uniforme sur [-2,2]
v=borne*npr.rand(N) # v =tirage de N valeurs suivant une loi uniforme sur [0,1/pi]
# On garde dans X tous les éléments de u vérifiant v(i)<sqrt(4*u(i)^2)/(2*pi)
X=u[(v<np.sqrt(4.-u**2)/(2*pi))]
temps_calcul_vectoriel = time()-temps_debut
# On récupère dans N_X la taille du vecteur X ainsi construit
N_X = np.size(X)
#densite de la loi de Wigner
x = np.linspace(-2., 2., 100) # discrétisation de l'intervalle [-2,2]
f_x = 1 / (2 * pi) * np.sqrt(4 - x ** 2) # discrétisation de la densité théorique de Wigner
# Affichage des resultats
plt.hist(X, normed=True, bins=round(np.sqrt(N_X)/10), label="Loi de Wigner empirique")
plt.plot(x, f_x, "r", label=u"Densité de la loi de Wigner : $f(x)=2*\pi/\sqrt{4-x^2}$")
plt.legend(loc='upper left' , bbox_to_anchor = (1., 1.))
plt.show()
print u"Temps de calcul de ", N_X, u" réalisations de la loi de Wigner, en vectoriel = ",
    temps_calcul_vectoriel
#
# Calcul via des boucles
#Nombre de tirages
N = N_X
#on genere dans X N_X realisations de la loi de Wigner par methode du rejet
#on comparera le temps de calcul avec des boucles avec le temps de calcul en vectoriel
temps_debut = time()
X = []
k = 0
while k < N:
   u = 4. * npr.rand() - 2.
   v = borne * npr.rand()
   while v > np.sqrt(4. - u ** 2) / (2 *pi):
      u = 4 * npr.rand() - 2
      v = borne * npr.rand()
   X.append(u)
   k += 1
   # fin du tant que
# fin de tant que
temps_calcul_boucle = time()-temps_debut
#densite de la loi de Wigner
x = np.linspace(-2., 2., 100)
f_x = 1 / (2 * pi) * np.sqrt(4 - x ** 2)
#
plt.hist(X, normed=True, bins=round(np.sqrt(N)/10), label="Loi de Wigner empirique")
plt.plot(x, f_x, "r", label=u"Densité de la loi de Wigner")
plt.legend(loc='lower right')
plt.show()
print u"Temps de calcul de ", N_X, u" réalisations de la loi de Wigner, via des boucles = ",
    temps_calcul_boucle
# Calcul du gain de temps
gain = (temps_calcul_boucle-temps_calcul_vectoriel)/temps_calcul_boucle*100
print "gain du vectoriel sur les boucles = ",gain,"%"
#Exemple de resultats
# Temps de calcul de 78604 réalisations de la loi de Wigner, en vectoriel = 0.00872087478638
# Temps de calcul de 78604 réalisations de la loi de Wigner, via des boucles = 0.418179035187
# gain du vectoriel sur les boucles = 97.9145595421 %
```



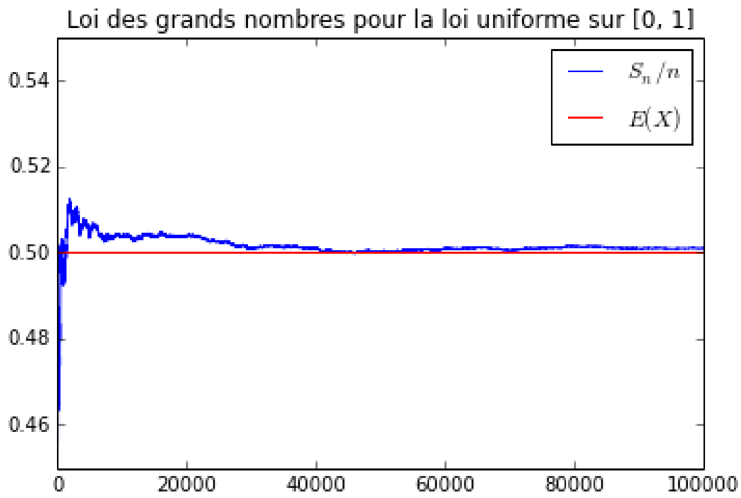
4. Loi des grands nombres

On rappelle que si (X_n) est une suite de variables aléatoires indepéndantes identiquement distribuées et ayant une espérance m, alors la suite de variables aléatoires

$$\bar{X}_n = \frac{X_1 + \cdots + X_n}{n}$$

converge presque sûrement vers m lorsque n tend vers l'infini. Simuler un grand nombre N de variables aléatoires de loi uniforme sur $[0, 1]$, tracer sur un même graphe la représentation de la suite $\bar{X}_n$ pour n variant de 1 à N et la courbe d'équation $y = m$, et observer cette convergence. On pourra utiliser la fonction `axhline(y)` qui trace une droite horizontale à la hauteur `y`. Refaire la même chose avec des lois gaussiennes.

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as npr
#
##############################################################################
## Loi des grands nombres pour la loi uniforme sur [0,1]
N = 10**5 # taille de l'echantillon
X = npr.rand(N) # tirage de N valeurs, stockees dans X, suivant la loi uniforme sur [0,1]
M = np.cumsum(X) / np.arange(1, N+1) # somme cumulee dans M des valeurs de X, divisee respectivement
    par 1, 2, 3, ... N
plt.figure()
p, = plt.plot(M)
a = plt.axhline(0.5, color="r") #tracer d'une droite horizontale passant en 0.5, milieu de [0,1]
plt.legend([p, a], ["$S_n / n$", "$E(X)$"])
plt.title("Loi des grands nombres pour la loi uniforme sur [0, 1]")
plt.axis([0,N,.45,.55])
#
##############################################################################
## Loi des grands nombres pour la loi normale de moyenne m et d'ecart type s
m=1
s=2
X = m+ s*npr.randn(N)
M = np.cumsum(X) / np.arange(1, N+1)
plt.figure()
p, = plt.plot(M)
a = plt.axhline(m, color="r") #tracer d'une droite horizontale passant en m, moyenne de la gaussienne
plt.legend([p, a], ["$S_n / n$", "$E(X)$"])
plt.title("Loi des grands nombres pour la loi gaussienne \n de moyenne $m =$ " +str(m) +u" et d'écart
    type $s =$ " +str(s))
plt.axis([0,N,.5,1.5])
plt.show()
```



5. Théorème central limite

On rappelle que si (X_n) est une suite de variables aléatoires indépendantes identiquement distribuées et de carré intégrable, d'espérance et écart-type communs notés respectivement m et s, alors en définissant

$$\bar{X}_n = \frac{X_1 + \cdots + X_n}{n},$$

la suite

$$\sqrt{n}(\bar{X}_n - m)/s$$

converge en loi vers la loi gaussienne standard lorsque n tend vers l'infini.

(1) Lorsque la loi des X_i est la loi uniforme sur $[-\sqrt{3}, \sqrt{3}]$, que valent m et s ? Visualiser, pour $n \gg 1$, la proximité de la loi de $\sqrt{n}(\bar{X}_n - m)/s$ avec la loi gaussienne standard (à l'aide d'un histogramme, en simulant un grand nombre de réalisations indépendantes de $\sqrt{n}(\bar{X}_n - m)/s$).

(2) Afin d'illustrer la différence de nature profonde entre la convergence presque sûre et la convergence en loi, tracer la représentation d'une réalisation de la suite $\sqrt{n}(\bar{X}_n - m)/s$, pour n variant de 1 à N. Cette suite semble-t-elle converger ?

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
import numpy.random as npr
import scipy.stats as sps

#Nombre de tirages de Monte Carlo pour calculer S_N
N = 100000
#Nombre de realisations de sqrt(N)S_N
n=1000
# Lois uniformes sur [-sqrt(3), sqrt(3) ]
X = 2 * np.sqrt(3) * (npr.rand(n,N) - 0.5) # Tirage dans la matrice X de (nxN) valeurs suivant une loi
     uniforme sur [-sqrt(3),sqrt(3)]
m = 0.
s = 1.
T = np.sqrt(N) * np.mean(X, axis=1) # Calcul dans T, des moyennes des lignes de la matrice X, divisees
     par sqrt(n)
borne = max(abs(min(T)),max(T)) # Calcul dans borne de la valeur absolue du max des valeurs de T
x = np.linspace(-borne, borne, 100) # discretisation dans x du segment [-borne,borne]
f_x = sps.norm.pdf(x, m, s) # calcul de la gaussienne standard en x
#
# Aficahge des resultats
#
plt.figure()
plt.hist(T, normed=True, bins=30, label="histogramme")
plt.plot(x, f_x, "r", label=u"Densité de la loi gaussienne standard")
plt.legend(loc='upper left' , bbox_to_anchor = (0., -0.1))
plt.title(u"Théorème central limite pour les lois uniformes sur [$-\sqrt{3}$, $\sqrt{3}$]")
#
#on recupere la 1ere trajectoire de X dans X1
```

```
#
X1 = X[0, :]
arange_N = np.arange(1,N+1)
T1 = np.sqrt(arange_N) * np.cumsum(X1) / arange_N # Calcul dans T1 des sommes cumulees de X1,
    normalisees respectivement par 1, 2, ...,N
#
# Affichage de T1
#
plt.figure()
plt.plot(T1)
plt.title("Evolution de $\sqrt{N}(\overline{X} - m)$ en fonction de N")
plt.show()
```